Firewall Policy Query Tool Whitepaper

Nichole Boscia[1]
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035-1000
Nichole.K.Boscia@nasa.gov

Introduction

Many enterprise organizations have various layers of security within their network, often at their border, across different branches and work groups. Firewalls are used to implement access restrictions and individual rules can quickly add up to the thousands. The increased complexity, along with massive volumes of layered rules, makes it very time consuming for network and security staff to determine end-to-end host accessibility. This paper describes how to create a firewall policy query tool that enables efficient discovery of firewall rules in large, complex, enterprise network environments, modeled after a tool that developed by the NASA Advanced Supercomputing Networks Team.

A typical firewall request would be to allow any host to communicate to a specific TCP port on a specified server, such as a web service. Without any tools, administrators would have to first determine all the firewalls that are impacted by this change by identifying source and destination networks, as well as any other networks used in transit. Then, the administrator must determine which firewall rule sets are applied to the previously determined interfaces, including both inbound and outbound traffic rules. While simply figuring out where changes need to be made can be quite time-consuming in itself, it also requires a major effort of reading through hundreds, or thousands, of individual rules that permit or deny traffic. A rule that blocks a service can be very difficult to locate, especially if it's implicitly defined. Even a seemingly simple request can end up taking hours to identify all the necessary pieces.

The firewall policy query tool presented in this paper aims to completely eliminate the manual component of this task. It automatically learns the network, firewall points, and individual rules, and then calculates paths and tells the administrator specifically which rules permit or deny a service and where. Additionally, any other staff can use the tool for troubleshooting connectivity issues without having to know how to read firewall rules. The tool facilitates network security by eliminating the human error factor and performing the needed rule logic inspections instantly.

Requirements

In order to automatically map network and firewall environments, the tool must be able to perform a number of different functions. Required functionality of this tool is to:

- Calculate the network path from source to destination within the local network, based upon routing topology learned from each device.
- Determine which routed network interfaces have firewall rules applied to them, based upon the path information.

[1] Employee of CSC, Inc. through contract no. NNA07CA29C with NASA Ames

- For each firewall rule set, calculate whether the traffic would be permitted or denied in inbound and outbound directions, based upon user-input data for source and destination hosts, port number, and protocol.
- Once the destination is reached, determine if the traffic is fully permitted end-to-end, based upon the calculations above.
- Allow users to submit queries through a simple-to-use, web-based user interface.

For the last of the above requirements, a web-based front-end that captures user input for source and destination hosts, port number, protocol type, and directional information was developed. When the user submits their information, the data is processed and output is displayed on the web page stating whether the traffic will be permitted or denied. Detailed information about each hop along the network, each firewall rule hit, and any matching rules are also displayed to aid in troubleshooting.

High-Level Design

To achieve the requirements outlined above, several components are needed. For the back-end, a database is needed to store routing and firewall data on which to perform queries, and a script is needed to collect data from the network devices. For the front-end, a web-based script will collect user input and submit queries to the database based on the specified parameters.

While there are multiple tools available to implement the design described here, this document defines specific requirements for the following software and hardware: MySQL database; Apache web server with PHP and DBI support; Linux system with SNMP, Perl, and Perl::DB; and Cisco routers with Access Control Lists and SNMP enabled. Specifications for these components are described below.

I. MySQL database

   The database server can reside anywhere on the network as long as it is reachable by the other components listed in this section. MySQL was chosen because it is well-supported, open-source software that will run well on the Linux servers already available. Explanations of tables needed for this tool are included in this document, but database administration is outside the scope of this project.

II. Apache web server with PHP and DBI support

   Again, a mature, open-source package was chosen here due to large community support and availability on the Linux servers. The web server must communicate to the MySQL database, or they can reside on the same system. The module for PHP support, mod_php, must be installed into Apache, and for PHP, the DBI module needs to be installed for the database API. This paper does not cover setting up a web server; it is assumed that one is already setup. Additionally, PHP is just an example of one possible language to use. It was chosen based on familiarity with the language and simplicity with database queries. Other web programming languages can be used, but their syntax is not covered here.

III. Linux system with SNMP, Perl, and Perl::DB

   Another system is needed to run the data collection scripts. It needs to be able to access the database and can be run on the same system, if desired, as long as it can reach all the network devices. Since the data is collected using the industry-standard protocol, SNMP, the SNMP client

software should be installed on the system, along with Perl, and Perl's database API, Perl::DB. These are all open-source, standard, software packages.

IV. Cisco routers with Access Control Lists and SNMP enabled

This paper covers only Cisco products. SNMP server must be enabled on these devices so that the collection script can pull routing data from them. Non-Cisco devices will most likely work as long as they support a management information base (MIB) that allows their routing table information to be read.

Project Components

The project design covers several areas that all must be integrated:

I. Collection of Firewall Rules
II. Firewall Database Design and Rule Separation
III. Import of Routing Information
IV. Routing Database Design and Insertion
V. Web Front-End and Database Query Logic
   A. Routing Logic
   B. Firewall Logic
VII. User Interface

I. Collection of Firewall Rules

The first step in this project is to obtain all firewall rules, the interfaces they are applied to, and the direction (inbound or outbound), in a simple and automated way. Unfortunately, there is no standard mechanism for retrieving any of this information in a way that is consistent with the SNMP method used throughout this project, because most vendors do not publish firewall rule details through SNMP. Alternative methods must be used in this case. There are three possible ways to accomplish this:

- Retrieve from existing backups
- Log in to the devices and show access lists
- Export from third-party firewall management software

Using configuration backup files provides a quick and easy way to access firewall rules and determine which interfaces they are applied to. No passwords have to be saved, making it a low-risk technique. One downside to this is that if backups are not done frequently, the information collected will be out-of-date and the output from the tool will be unreliable. However, if firewall changes are not frequently made, or if backups are done throughout the day, then this method should suffice. Another downside to this method is that sequence numbers are not typically stored in the configuration file. This means that virtual sequence numbers would have to be associated with each rule so their order in the database is maintained during queries. While the virtual sequence numbers may not match exactly what the firewall shows in its run-time rule lists, the tool will function as long as the order is identical.

Another method is to use an automated login session to your devices. Once logged in, commands are run to show and capture all firewall rules and interface configurations. This will guarantee the most accurate and up-to-date data collection. Additionally, sequence numbers will match what the device uses. As long as this technique can be secured in a way that ensures saved passwords are encrypted and access is

restricted to meet local security requirements, this is the most direct approach. This is the optimal approach for the environment in which the tool was designed to run for this project.

Alternatively, if firewall rules are already being imported from some third-party software, you may be able to retrieve and incorporate them into the project database. However, since every product is different, it is up to the developer to determine how to retrieve the necessary information and whether or not it can be automated. Usually, there is a way to export rules to a CSV or XML file, or even a back-end database that can be accessed directly.

The objective here is to have a reliable way to obtain current firewall rules, and where those rules are applied. There is no standard method for this, and it is highly dependent on your existing administrative techniques. Once that information is gathered in a reliable and automated way, the foundation for building the database comes next.

II. Firewall Database Design and Rule Separation

When performing complex queries, the rules must be strategically entered into a database. The table structures are key for providing quick response times, especially for complex, large-scale networks. There will be multiple tables that cover routing, firewall rules, and device information. These will all be cross-referenced to perform intricate queries used in determining network paths and firewall logic.

To start, a simple table that maps network devices to a unique identifier, such as Table 1, is created.

Table 1

Field	Type	Null	Key	Default	Extra
deviceID	smallint(5) unsigned	YES	PRI	NULL	
deviceName	varchar(25)	YES		NULL	

For example, your table should contain all firewall devices and look somewhat like Table 2.

Table 2

deviceID	deviceName	type
1	border	NULL
2	core-0	NULL
3	core-1	NULL
4	dmz-router	NULL

Once this table is in place and populated with your devices, it is time to map which firewall rule sets are applied to various interfaces on each device, as shown in Table 3.

Table 3

Field	Type	Null	Key	Default	Extra
aclID	smallint(5) unsigned		PRI	NULL	auto_increment
deviceID	smallint(5) unsigned	YES	MUL	NULL	
name	varchar(35)	YES		NULL	
interface	varchar(128)	YES		NULL	
description	varchar(200)	YES		NULL	
lastModification	datetime	YES		NULL	

Here, each firewall rule set is associated with a unique ID. This is important because the same access list can exist across multiple devices, usually for redundancy or disaster recovery designs. The second column maps to the deviceID as previously defined in Table 2. The name column is simply to define the name of the firewall rule set, which is used in troubleshooting or for increased output detail to the user. The interface fields are key here—they map a specific rule set to an actual interface, which will be referred to once we start pulling down routing information**.** Alternatively, the actual unique interface ID (*ifIndex*) may be used instead of the description string for optimization. The last two columns, description and lastModification, are used to enable better documentation, but are not required for basic functionality. Table 4 captures a snippet of what your data should look like once entered.

Table 4

aclID	deviceID	name	interface	description	lastModification
1	3	financial_in	Vlan100	finance systems	2008-11-25 12:52:07
2	3	financial_out	Vlan100	finance systems	2008-09-29 19:40:05
3	2	servers_in	Vlan200	internal servers	2009-04-30 11:37:55
4	2	servers_out	Vlan200	internal servers	2009-10-15 16:52:38

Another mapping table, such as Table 5, needs to be created to associate rules to specific rule sets and to define sequence numbers, which are required for properly ordering rules when making a query. This is a straightforward mapping of uniquely assigned rules being given a ruleID and mapped to a rule set (aclID) in Table 4. The column for sequence numbers is information that was obtained back in Section I, and can be actual numbers taken from the firewall rule output, or self-assigned sequential values, as long as the rules can be properly ordered since they are processed serially. After some input, your mapping of individual rules to aclIDs should look like Table 6.

Table 5

Field	Type	Null	Key	Default	Extra
ruleID	mediumint(8) unsigned		PRI	NULL	auto_increment
aclID	smallint(6) unsigned	YES	MUL	NULL	
seqNum	mediumint(8) unsigned	YES		NULL	

Table 6

ruleID	aclID	seqNum
1	3	100
2	3	110
3	3	120
4	2	100

The next part is the most complicated thus far. In Section I, all rules were collected through an automatic mechanism. Once those rules are retrieved, they need to be inserted into the database. To do this, a new table is created and breaks each rule down into its components (see Table 7).

Table 7

Field	Type	Null	Key	Default	Extra
ruleID	mediumint(8) unsigned		PRI	0	
action	varchar(15)	YES		NULL	
proto	varchar(10)	YES		NULL	
srcip	varchar(15)	YES		NULL	
srcmask	varchar(15)	YES		NULL	
src_qual	varchar(5)	YES		NULL	
src_start	smallint(5) unsigned	YES	MUL	NULL	
src_end	smallint(5) unsigned	YES	MUL	NULL	
dstip	varchar(15)	YES		NULL	
dstmask	varchar(15)	YES		NULL	
dst_qual	varchar(5)	YES		NULL	
dst_start	smallint(5) unsigned	YES	MUL	NULL	
dst_end	smallint(5) unsigned	YES	MUL	NULL	
flags	varchar(25)	YES		NULL	

There are many different ways to write firewall rules, and they vary from vendor to vendor, but the same general information is required. A typical firewall rule follows the syntax: *action protocol source_ip source_mask qualifier port(s) destination_ip destination_mask qualifier ports(s) flags*. For example:

```
permit tcp 1.1.1.0  0.0.0.255  2.2.2.0 0.0.0.255 eq 80
deny   udp 1.1.1.1  0.0.0.0    0.0.0.0 0.0.0.0   eq 22
```

The ruleID for each rule is defined in Table 7. The action is typically “permit” or “deny.” Here, this variable is set to a string for comprehension, but a more efficient definition would be to enumerate the values, or use a straight integer representation. The protocol column is similar to the action one—a string is used here, but enumeration is more efficient if you have a large number of rules to query. Examples of protocols are IP, TCP, UDP, and ICMP. The last column for flags is to match on TCP flags such as SYN, ACK, and FIN. This is often used to determine if a connection is being initiated or if it is a response.

Data for the source and destination IP addresses and wildcard mask are important here for bitwise *OR* calculations to determine if an address matches within a range. Notice that a wildcard mask is used and not a subnet mask, as is standard for firewall rule syntax. This means general rules matching “any” would be translated to an IP address of 0.0.0.0 and a wildcard mask of 255.255.255.255. It is recommended that you use integer formats for the IP addresses using such functions as *ip2long* to convert the strong. This optimizes queries if you have a large set of rules. For purposes of documentation and examples, these are entered here as strings.

The last piece of data is the port information. The qualifier column is to define whether a port is being matched to a single port number (“eq”) or to a range of ports using qualifiers such as “gt,” “lt,” or “range.” For the starting port, if only one port is being matched, the port number is entered here; otherwise, the first port of a range is entered. No end-port needs to be defined (NULL) for a single port, or if a qualifier such as “gt” or “lt” is used. Only a specifically defined range such as “100–1000” requires the end port to be defined. If desired, a hard boundary such as “0” or “65535” can be entered, or the value range can be restricted on the front-end to legitimate values.

This will be the largest, most complex table for all the firewall rules. Data must be translated and entered precisely, as in Table 8, for the bitwise calculations to work. At this point, all the information that needs to be entered for your firewall rules is complete and it is time to get into the routing details to determine the path a packet takes through your network.

Table 8

ruleID	action	proto	srcip	srcmask	src_qual	src_bgn	src_end	dstip	dstmask	dst_qual	dst_bgn	dst_end	flags
1	permit	udp	1.1.1.1	0.0.0.0	NULL	NULL	NULL	2.2.2.2	0.0.0.0	range	5580	5591	NULL
2	permit	tcp	2.2.2.0	0.0.0.255	NULL	NULL	NULL	3.3.3.20	0.0.0.0	eq	22	NULL	NULL
3	deny	udp	2.2.2.2	0.0.0.255	NULL	NULL	NULL	1.1.1.1	0.0.0.0	gt	1024	NULL	NULL
4	permit	tcp	3.2.1.0	0.0.0.255	eq	555	NULL	1.2.3.4	0.0.0.0	lt	1024	NULL	syn

III. Import of Routing Information

Capturing the local network's routing table allows simulation of a packet's route through the network to determine which firewalls are traversed. The network is often dynamic and routing changes occur, impacting which firewalls are in the path. With highly redundant networks, there are often multiple firewalls for a single network, which can get out of sync as well. Being able to identify exactly which devices a packet gets sent through helps with troubleshooting access issues.

There are several ways to import your routing information. The most intrusive way is to have an automated SSH session log in to each device and grab the output when showing the routing table, but you will also have to parse through the configuration to determine which routes exist on local interfaces and which are learned from routing peers. Another technique is to pull routing information remotely through SNMP. This is easier to do from a programming perspective and also works for mapping out additional information such as routing protocol. A third method, which is the most dynamic, would be to have a system run a local routing protocol, such as OSPF, and passively join the area, thus instantly retrieving every update passed between routers. At pre-determined intervals, the routing information would need to be dumped and inserted into the database. The limitation of this method is that there can be aggregated routes advertised to the system you are listening on, or even non-captured private routing for VPNs, since data is not being pulled directly from other routers.

For this project, the SNMP technique is used because it is the most thorough and robust. Any version of SNMP can be used, as long as there is read access to the proper OIDs. The MIBs that are supported vary between vendors and devices, but what is used here is fairly standard. For programming purposes, Perl is used to quickly parse through the output and insert it into the MySQL back-end.

The first action is to connect to your database where all devices were defined, and pull that list down. For each one of those devices, routing information will need to be collected. To collect the basic next-hop address for each of your local subnets, query the *ipRouteNextHop* OID. A list of networks will be returned in the form of:

```
RFC1213-MIB::ipRouteNextHop.1.1.10.0 = IpAddress: 1.1.10.1
RFC1213-MIB::ipRouteNextHop.1.9.10.0 = IpAddress: 1.9.10.254
RFC1213-MIB::ipRouteNextHop.0.0.0.0 = IpAddress: 9.9.9.1
```

This gives you all the subnets that the device being queried knows about, including where it forwards a packet in order to reach a destination IP address. If you are querying any border routers, you will want to exclude networks that are learned through exterior routing protocols such as BGP, unless they are within your organization's firewalls.

The next step is to determine the routing protocol type. This will distinguish between routes learned though a peer and those that exist locally. This is done using the *ipRouteProto* OID. Output looks like:

```
RFC1213-MIB::ipRouteProto.1.1.10.0 = INTEGER: local(2)
RFC1213-MIB::ipRouteProto.1.9.10.0 = INTEGER: ospf(13)
RFC1213-MIB::ipRouteProto.0.0.0.0 = INTEGER: ospf(13)
```

All that needs to be captured here is a mapping of the subnet to the type. If the response is "local," then that device has the last hop in the path; otherwise, it is forwarding it on to another device. The actual numbers following the protocol do not need capturing, but the protocols will need to be enumerated when entered into the database.

Now it is time to identify the subnet masks of the local subnets to add that data to the information already captured. Again, a standard OID called *ipRouteMask* is used, which gives output similar to the previous OIDs:

```
RFC1213-MIB::ipRouteMask.1.1.10.0 = IpAddress: 255.255.255.252
RFC1213-MIB::ipRouteMask.1.9.10.0 = IpAddress: 255.255.255.0
RFC1213-MIB::ipRouteMask.0.0.0.0 = IpAddress: 0.0.0.0
```

Given that the subnet is listed in each of these responses, there should be a clear mapping from a single subnet to its mask, next hop, and type, for each device. Using the next-hop information obtained in the first SNMP query (*ipRouteNextHop*), the actual unique interface identifier (*ifIndex*) can be retrieved for subnets local to the device. In the output from the *ipRouteProto* query, one of the routes is local. If that subnet is queried using the *ipAdEntIfIndex* OID, the actual interface ID will be returned. This will tell us exactly which interface the packet will get routed through so that it can be checked for firewall rules. The actual query requires the subnet to be prepended to it, such as:

```
[shell]$ snmpget -v1 -c public deviceName ipAdEntIfIndex.1.1.10.0
IP-MIB::ipAdEntIfIndex.1.1.10.0 = INTEGER: 93
```

It is important to note that most devices are able to have persistent *ifIndex* mappings, so that if new ports are added, the actual interface IDs do not change. This feature typically has to be enabled though an SNMP command on the device. It is not required, but keeps the information obtained valid for longer, unless automated updates are run frequently.

With the unique *ifIndex* value retrieved, all that remains is to grab its interface description. This will map back to the interface information, which was pulled from the configuration files and maps specific firewall rule sets to their attached interface. Using a simple *ifDescr* query with the *ifIndex* value appended, we can retrieve the interface description:

```
[shell]$ snmpget -v1 -c public deviceName ifDescr.93
IF-MIB::ifDescr.93 = STRING: "Vlan100"
```

The information collected so far for each device includes all subnets learned, their next hop, their subnet mask, their type, their unique *ifIndex*, and the device's configured description for local routed interfaces. For most organizations, this is enough to get started on entering everything into a database. However, many sites have virtual private networks (VPNs), which add complexity to the design. For these cases, a procedure for gathering the same information is included.

Optional VPN Procedure

This section desribes how to obtain routing information for Virtual Route Forwarding (VRF) used on Cisco devices. This includes uses for MPLS and tunnels. The MIBs are likely specific to Cisco, so if another vendor is used, the appropriate MIBs will need to be identified. Also, this does not cover every possible VPN implementation. Once the information is retrieved, it will be entered into the database with the other routes learned.

As before, the first thing to do is get a listing of all the subnets. Thankfully, there is an OID that gives not only the list of private subnets, but the VRF instance ID (also known as the Route Distinguisher) to which they are attached, as well as their subnet mask and the next hop, all in a single query. The output is tricky though, and will be described in sections:

```
[shell]$ snmpwalk -v1 -c public deviceName SNMPv2-SMI-v1::experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45

SNMPv2-SMI-v1::experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45.65.4.10.1.2.0.4.255.255.255.0.0.4.0.0.0.0 = Gauge32: 0
SNMPv2-SMI-v1::experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45.65.4.10.1.4.0.4.255.255.255.0.0.4.0.0.0.0 = Gauge32: 0
SNMPv2-SMI-v1::experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45.66.4.0.0.0.0.4.0.0.0.0.0.4.10.2.1.254 = Gauge32: 0
SNMPv2-SMI-v1::experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45.66.4.10.2.1.0.4.255.255.255.0.0.4.0.0.0.0 = Gauge32: 0
```

The first part of the output, `(experimental.118.1.4.1.1.5.9.99.111.110.115.111.108.101.45)`, is simply the OID that you queried. Following that is the unique identifier for the VRF instance. In this case, there are two: 65 and 66. This means that there are two VPNs on the device. Consider these numbers as similar to the *ipRouteProto*, because they will be entered in the same database column. The next number in the output is always "4," which acts as a separator. Following this is the subnet, followed by the "4" separator again, and then the netmask, followed by a "0" and the "4" separator, and then the next-hop. Breaking the first response above down into fields gives the following:

VRF ID: 65
Subnet: 10.1.2.0
Netmask: 255.255.255.0
Next-Hop: 0.0.0.0 (default)

Retrieving the unique *ifIndex* requires a complicated query using all data just obtained. The SNMP query is in the format of:

```
SNMPv2-SMI-v1::experimental.118.1.4.1.1.8.9.99.111.110.115.111.108.101.45.[vrf].4.[subnet].4.[netmask].0.4.[next-hop]
```

Inserting the values from the last query, we get:

```
SNMPv2-SMI-v1::experimental.118.1.4.1.1.8.9.99.111.110.115.111.108.101.45.65.4.10.1.2.0.4.255.255.255.0.0.4.0.0.0.0
```

Sending this query will give the *ifIndex* value (278 in this case) for that particular VPN route:

```
SNMPv2-SMI-v1::experimental.118.1.4.1.1.8.9.99.111.110.115.111.108.101.45.65.4.10.1.2.0.4.255.255.255.0.0.4.0.0.0.0 = INTEGER: 278
```

If a local interface does not exist, the integer "0" is returned as a result. For returned IDs, the interface description needs to be pulled to map back to the applied firewall interfaces. Just as before, this is done with a simple *ifDescr* query with the interface ID appended:

```
IF-MIB::ifDescr.278 = STRING: "Vlan200"
```

One last piece of information to gather is for all the routes that do not have a local route and were listed to go out the default route, as in our example (0.0.0.0). The actual interface the packet gets routed out of needs to be captured since there may be firewall rules there. The *ipNetToMediaType* OID is used for this, with the interface ID appended to it:

```
IP-MIB::ipNetToMediaType.278.10.1.2.17 = INTEGER: dynamic(3)
IP-MIB::ipNetToMediaType.278.10.1.2.18 = INTEGER: dynamic(3)
IP-MIB::ipNetToMediaType.278.10.1.2.254 = INTEGER: static(4)
```

The output will list all the host IP addresses on that subnet as dynamic, and the router interface as static. The static address needs capturing to be listed as the *ifIndex* value for that subnet.

Collecting all network routing information can take some time, depending on the size of your network. For fairly static networks, data can be gathered once a day, or as often as changes normally occur. The actual SNMP queries are done serially, so there is not much load on the network and device CPUs. It is now time to discuss what to do with all this information.

IV. Routing Database Design and Insertion

All routing information collected must get entered into a table that efficiently maps subnets to routes to firewall rules, while maintaining compartmentalization for VPN networks (see Table 9). The schema used is simple and similar to an actual routing table, except for the addition of defining firewall policies attached to routed interfaces.

Table 9

Field	Type	Null	Key	Default	Extra
id	int(11)		PRI	NULL	auto_increment
deviceID	int(11)	YES		NULL	
type	int(11)	YES		NULL	
subnet	varchar(15)	YES		NULL	
netmask	varchar(15)	YES		NULL	
nexthop	varchar(15)	YES		NULL	
aclID_in	int(11)	YES		NULL	
aclID_out	int(11)	YES		NULL	
ifIndex	int(11)	YES		NULL	
ifDescr	varchar(25)	YES		NULL	

As with all our other tables, Table 9 has a unique ID key field set to primary key. The *deviceID* refers to the unique ID given to each network device as shown in Table 1. The column *type* will contain the data collected using *ipRouteProto,* where the returned value was either “ospf” or “local.” The VRF ID (Routing Distinguisher) integer from the VPN section is also defined here. For optimization, the values in this column are set to integers, with “ospf” assigned a value of 2, “local” assigned a value of 1, and each instance of VRF is assigned its actual Routing Distinguisher value.

The *subnet*, *netmask,* and *nexthop* columns are straightforward. However, while they are listed here as a string to make the information easy to read, it is optimal to store them in unsigned integer format using the *ip2long* conversation as described in Section III.

Skipping the next two columns for now, the *ifindex* and *ifDescr* columns are also straightforward. The *ifDescr* column is not needed if the actual *ifIndex* value was used in the interface field when the rule sets were mapped to the interfaces. However, both are used here to easily display the output. Otherwise, the translation would have to be done in another table.

A little bit of extra work has to be done to obtain the values for the *aclID_in* and *aclID_out* columns. Back in Section I, the router configuration files were parsed to map out which interfaces had firewall rules applied, the names of the rules, and the direction for which they were applied. Then, in Section II, the names of the rules and their interfaces were entered into Table 4. So, to obtain the unique ID for the firewall rule set, a MySQL query must be made to select the aclID that was defined in Table 4. Using the sample data originally given, the query would look like this:

```
SELECT aclID from aclDetails where deviceID=”core-1” and name=”financial_in”;
```

There will only be one match found since the rule set has a unique name. The selection query needs to be performed for each rule set found to map it to its aclID and applied interface. That information is most easily entered *after* all other routing information is populated because it can be done through an update statement where the device and interface can be specified:

```
UPDATE routing set aclID_in=1 where deviceID="core-1" and ifDescr like "Vlan100";
UPDATE routing set aclID_out=2 where deviceID="core-1" and ifDescr like "Vlan100";
```

Once all the information is entered, you will have a full table that looks similar to Table 10.

Table 10

id	deviceID	type	subnet	netmask	nexthop	aclID_in	aclID_out	ifIndex	ifDescr
1	1	2	1.1.10.0	255.255.255.0	1.1.10.1	NULL	NULL	NULL	
2	2	1	1.9.10.0	255.255.255.0	1.9.10.254	1	2	300	Vlan100
3	8	2	0.0.0.0	0.0.0.0	9.9.9.1	NULL	NULL	NULL	
4	3	66	10.1.10.0	255.255.255.0	10.2.11.254	3	NULL	358	Vlan200
5	4	65	10.2.5.0	255.255.255.0	10.1.50.254	NULL	NULL	352	Vlan300

Using this data, along with the firewall tables we entered earlier, one can fully simulate a packet traversing the local network. This requires knowing the logic a router must use to process each packet, as discussed in the next section.

V. Web Front-End and Database Query Logic

To make use of the database, a front-end interface must be implemented. The most accessible front-end is a simple website that allows users to enter specific information, such as source and destination IP address, and use those values to query the database via a scripting language.

The minimum requirement for end-to-end policy testing is a source and destination address, which would imply IP or ICMP protocols instead of a higher-layer protocol such as TCP or UDP. The next level of complexity is adding TCP/UDP as a protocol and defining a specific port. Lastly, there is also a checkbox option for uni-directional UDP, since not all UDP flows are bi-directional. A web form with PHP processing is used to collect this information from the user and to validate the entries (ensuring a port is defined for TCP/UDP queries, IP address formats are correct, etc.).

The first thing to get out of the way is loading a key/value pair for the deviceID to deviceName (Table 2) and also the unique aclID to its actual name (Table 4). This is mostly for output purposes when displaying to the user. The next part is to initialize all variables to be used. Most are used for keeping track of returned values and ensuring the queries do not get stuck in an infinite loop.

```
$oneWay = 0;  // Set to "1" if the uni-directional checkbox is checked.
$srcMatch = 0; // Set to "1" if the source IP address resides within the local network
$dstMatch = 0; // Set to "1" if the destination IP address resides within the local network
$loop = 0; //  Set to "1" once the end has been identified
$badValue = 0; // If some values are determined to be invalid, this is set
$status = 0; //  Return value of firewall query function. Set to "1" if packet is permitted through firewall.
$revStatus = 0; // Same as above, except for the opposite direction
$denied = 0; // Set to "1" if a packet is denied.
$revDenied = 0; // Set to "1" if a packet is denied in the reverse direction.
$borderRtr = 10; // This is the deviceID of your border router. Required to know the point of entry/exit.
```

For simplicity of documentation, the routing and firewall logic are going to be separated. They are two distinct processes, so it is easier to separate them and detail each one.

A. Routing Logic

Now for the actual route processing. First one needs to determine whether the given source IP address is local or external to your network (such as google.com). If it is local, then the aclID of its subnet interface will be retrieved. To begin, some bitwise AND math needs to be done between the subnet and subnet mask:

```
select * from routing where type!=2 and (inet_aton(subnet) & inet_aton(netmask) =
inet_aton("[source IP]") & inet_aton(netmask)) and ifIndex!=0 limit 1;
```

We are only interested in where the subnet actually has an interface. In our routing table, the *type* field was set to "1" for local routes, "2" for routes learned via a routing protocol such as OSPF, and the actual VRF Route Distinguisher value is used for VPNs. This means that we ignore the routes learned via a routing protocol. For VPN routing, the *ifIndex* may get set to "0" depending on how the vendor implemented the MIB, so those instances should also be ignored if VPNs are involved. All columns are selected for output purposes, to display the path to the user.

If a row is returned, that means the source IP address is local. In the row that is returned, if either aclID_in or aclID_out have a value assigned (i.e., not NULL), the function for querying the firewall rule will be called and all relevant parameters will be passed (see Section B). This is to be done for both directions. The only change in the second direction is specifying the direction flag when calling the function. If the return value for these cases is "0," then the packet would be blocked and the tracking variables $denied and $revDenied would be set to "1." It is important to keep in mind that both directions have to be checked for bi-directional traffic. It only takes one "deny" rule to block the entire flow. The exception to this is if the flow was defined to be UDP uni-directional, which is an option in the front-end interface. If flow is only in a single direction, then the outbound rule set can be skipped since it is never hit. This method is used for every call to the firewall function, so keep this information in mind.

For the matched row, the *nexthop* ($nextHop), *deviceID* ($srcDev), and *type* ($type) are saved to variables for future queries. If there were no matches for the query, then the address is either not local, or is invalid. The easiest way to check this is to see if it falls within your organization's address space. If it does, an error is given stating that the subnet is not currently in use. If it is determined to be an external address, some values must be defined to make it appear to come from "outside," through the border router. The firewall rules can be identified using the *deviceID* of the border router, which was defined in the initialization section:

```
select aclID_in,aclID_out from routing where deviceID=$borderID and  (inet_aton(subnet) &
inet_aton(netmask) = inet_aton((select nexthop from newRouting where subnet="0.0.0.0" and
deviceID=$borderID limit 1)) & inet_aton(netmask)) and type=1;
```

Once any rule set IDs are retrieved, they are passed to the firewall query function for packet processing, and the $denied and $revDenied variables are set as described earlier.

Similarly to the source address, the destination address must also be identified as local or external to the organization's network. This is done exactly the same way as with the source IP address, except with the destination IP address substituted. This time however, there is no need to check the firewall rules, since that will be done automatically when processing the routing path. If the destination address is within the local network, the returned values for the subnet and interface need to be saved to the variables $dstNet and $dstInt for future queries and identifying when the destination has been reached. If the destination is some external address, the variable $dstNet is set to "0.0.0.0" to indicate the default path out, and $dstInt is set to NULL, since the actual destination interface does not matter (outside of the network).

At this point, a quick check should be done to determine if both the source and destination IP addresses are external to your network. If neither of the IP addresses falls within subnets existing in the database, then the traffic will be completely outside of your network and will never hit your firewall. An error message should be displayed stating that at least one IP address needs to be within your organization.

When a router gets a packet, it looks at the destination address and determines if it matches any of the subnets it has learned in its routing table. If there is a match, it finds which interface is defined as the next hop (or if it is locally attached), and forwards it out the interface that matches the next-hop subnet. If there is no match, it sends it to the default route as a next hop. This procedure continues until the next hop is itself on a local interface. Using that methodology, a programming loop is made to continuously perform the same set of queries through each device in the path until the destination is reached and the loop is exited.

Inside the loop (*while $loop == 0*), the first step is to find the local subnet interface of the next hop to see if firewall rules exist there. This is the interface going *out* of the router to the next hop. Using variables that were assigned values in the last query, this can be achieved as follows:

```
select * from routing where deviceID=$srcDev and (inet_aton(subnet) & inet_aton(netmask) =
inet_aton((select nexthop from routing where subnet="$dstNet" and deviceID=$srcDev limit 1)) &
inet_aton(netmask)) and type=$type;
```

The reason for keeping track of *type* here is mostly for VPNs. We only want routes returned within the same VPN. If no VPNs exist, the *type* would always be "1" here. The nestled select statement returns the IP address of the next hop for the destination subnet on the device that is the gateway for the source IP address. Using that returned value, the main select finds which local interface that packet will be forwarded out of to get to the next hop.

Using the returned information, some checks need to be done. Compare the $dstInt variable defined earlier to the one returned by the select. If they match, set a flag to break the loop (such as setting *$loop = 1*) because the destination is reached. Next, if the columns *aclID_in* or *aclID_out* exist, the firewall check function must be called. However, because the traffic is flowing outbound here, the actual source and destination information need to be swapped. Consider this the reverse flow, where the packet headers contain the destination as the source. This is easily accommodated by swapping the direction flag in the verify function (see Section B). One more field to note is the unique ID of the row returned. This ID needs to be saved for comparison with the next unique row ID returned while inside the loop process.

One more optional check to perform is again related to VPNs. If the returned *nexthop* value is that of a Network Address Translation (NAT) device, then the source IP address should be changed to the public IP address. This has to be done manually since routing protocols are not aware of address translations. It is a minor edit to provide additional functionality. If no matches are returned, then there is no end-to-end path between the source and destination IP addresses and an error message is displayed.

While the loop is still going ($loop hasn't been set to "1"), the next step is to locate the next device in the path and gather its subnet information:

```
select * from routing where nexthop=(select nexthop from routing where subnet="$dstNet" and
deviceID=$srcDev  limit 1) and type=$type;
```

This returns the entire row matching the IP address of the next hop from the device that was queried before. The path has progressed one hop. The first thing to check for is whether the unique ID of the

returned row matches of the previous query. If so, then this is a routing loop and $loop should be set to "1" to break it. Otherwise, the same check is performed for aclID_in and aclID_out as originally specified. This is a normal flow, not a reverse one, so direction does not need to be swapped.

To continue the loop logic, the variables $srcDev and $subnet are now set to the row just returned. However, if no row was returned, that means there is no device on the local network and the packet has reached the border or some other end device, so the loop should be set to break.

At this point, if the loop variable is "1," then the end has been reached and results of the firewall queries should be posted. Otherwise, the loop starts over and the query for finding the outbound interface of the source device is checked, followed again by finding the device of the next hop. Eventually, the end device will be reached.

B. Firewall Logic

As described in Section A, each time a routed interface is reached, a firewall check is performed. The function prototype used is:

```
function verify(database_ptr, aclID, source_IP, dest_IP, source_port, dest_port, protocol, direction)
```

The database pointer is the variable to which the database connection is assigned. The aclID parameter is the unique rule-set ID obtained from *aclID_in* and *aclID_out* in the routing table. Next are the source and destination IP addresses, followed by their respective ports. The protocol could be IP, ICMP, TCP, or UDP, as defined in the front-end options. Direction is important here, because it determines if the rule sets are being checked for a forward or reverse flow. This does not directly correlate to inbound and outbound because a flow goes in one port of a router and out another. The reverse flow does the same. The direction of the flow needs to be tracked during the routing logic and passed along to this function.

If the directional flag is set to "1," then the flow is reversed and the variables for source and destination IP addresses and ports need to be swapped. The actual query is broken up into pieces, which depend on whether or not variables have values set. The first section of the query begins with:

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID="$aclid" and a.ruleID = b.ruleId;
```

The ruleDetails table (Table 6) is where each rule was assigned a unique ID, mapped to an aclID, and included the sequence number of that rule. The ruleFields table (Table 8) is where every parameter of the firewall rule was separately entered into a table. This query simply finds all the rule parameters that match the aclID passed to the function.

If the destination IP is not defined, add the default (any) address to the query; otherwise, use the actual IP address given:

```
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("0.0.0.0")  | inet_aton(dstmask))
```

or

```
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("$dstip")  | inet_aton(dstmask))
```

The same thing is done with the source IP address:

```
and (inet_aton(srcip) | inet_aton(srcmask) = inet_aton("0.0.0.0") | inet_aton(srcmask))
```

or

```
and (inet_aton(srcip) | inet_aton(srcmask) = inet_aton("$srcip") | inet_aton(srcmask))
```

For ICMP protocol queries, the last piece of the query checks for the explicit ICMP or IP protocol type, since ICMP falls under the IP meta type:

```
and (proto="icmp" or protocol="ip") and flags is NULL  order by seqNum limit 1;
```

Ordering by sequence number ensures that the proper order is maintained, because even if there is a deny rule after a permit rule, the packet will have already been permitted. Because of this, the query is limited to only one rule returned.

A similar technique is used for simple IP protocol queries:

```
and proto="ip" and flags is NULL order by seqNum limit 1;
```

This is the simplest query. Now, each section should be concatenated for the full database query. For example, consider a query with the following values passed to the verify function:

```
verify($DB, 100, "1.1.1.1", "2.2.2.2", undefined, undefined, "ip", 0)
```

For this case, the actual query would look like:

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID=100 and a.ruleID = b.ruleId
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("2.2.2.2")  | inet_aton(dstmask)) and
(inet_aton(srcip) | inet_aton(srcmask) = inet_aton("1.1.1.1") | inet_aton(srcmask)) and proto="ip"
and flags is NULL order by seqNum limit 1;
```

The TCP and UDP queries are more complicated. Not only is port information introduced, but state and flow type also come into consideration. With TCP, a packet that has a "SYN" flag-only set is considered to be initiating a session. The response packet will have "SYN" and "ACK" flags set, and all future packets will have the "ACK" flag set as well. To the firewall, this is considered stateful logic, meaning it internally keeps track of whether the packet is initiating a connection, or if the packet is in response to an initiation request. Depending on the vendor, other factors come into play such as tracking sequence numbers, but since that is an internal check it does not need to be included here. Another concern is with UDP where flows can be either bi-directional or uni-directional. However, this was addressed back in the routing section where the firewall check function was skipped for outbound interfaces in the case where uni-directional UDP was defined.

For either UDP or TCP queries in the forward direction (where the source initiates to the destination), the most important piece is to verify that the port is allowed. This has to take into account not only exact port matches, but also range statements, as well as "greater than" and "less than" ranges. Also, the lower-layer protocol IP will match any TCP or UDP protocol type as well:

```
and (proto="ip" or (proto="$proto" and ((dst_qual="eq" and dst_bgn=$dstport) or (dst_end="gt" and
dst_bgn <= $dstport) or (dst_qual="lt" and dst_bgn >= $dstport ) or (dst_qual = "range" and
$dstport between dst_bgn and dst_end)) )) and flags is NULL order by seqNum limit 1;
```

The variables here are taken directly from the function call for protocol and destination port. The *flags* field (which lists whether the rule applies for established connections) is set to NULL because it should not match on established connections. This statement covers a basic IP protocol match (which does not even require port information), as well as the various options for port ranges.

The TCP protocol is unique in its stateful connection type. For the response flow, the source port is unknown, so rules have to be matched on state or on the destination port (which is the original source port). This is the most complex section of the firewall query as there are a lot of options and combinations to consider:

```
and (proto="ip" or (proto="$proto" and ((src_qual="eq" and src_bgn=$srcport) or (src_qual="gt" and
src_bgn <= $srcport) or (src_qual="lt" and src_bgn >= $srcport ) or (src_qual = "range" and
$srcport between src_bgn and src_end) or (dst_qual = "gt" and flags like "%estab%") or (src_qual
is NULL and dst_qual is NULL and flags like "%estab%") or (dst_qual = "gt" and dst_bgn <=
$srcport) or (dst_qual = "lt" and flags like "%estab%")) ))  and ((flags not like "%frag%") or
(flags is NULL)) order by seqNum limit 1;
```

This is similar to the forward flow query, only extended to cover established rules. Also, a statement should be included to prevent fragmented packets (packets that do not include a complete header) from being matched here. If your firewall automatically blocks them, however, then that section is not needed.

One last component option is needed for reverse UDP flows (those that are bi-directional). This is similar to the above TCP section except that there is no state maintained with UDP, so the destination port is swapped with the source port:

```
and (proto=”ip" or (proto="$proto" and ((src_qual="eq" and src_bgn=$srcport) or (src_qual="gt" and
src_bgn <= $srcport) or (src_qual="lt" and src_bgn >= $srcport ) or (src_qual = "range" and
$srcport between src_bgn and src_end) or (dst_qual = "gt" and dst_bgn <= $srcport) or (dst_qual =
"gt" and dst_bgn <= 1024)) ))  and ((flags not like "%frag%") or (flags is NULL)) order by seqNum
limit 1;
```

Since there is no state, and the destination system will typically allocate a port above 1023 (unless manually configured to do otherwise), a check is added to see if a rule permits UDP on a high-numbered port inbound to the destination (originating source).

To show more clearly how these components work, some examples of different scenarios are provided below to demonstrate what the combined SQL queries will look like.

Example 1

Source IP address: 99.99.99.99 (external to site)
Destination IP address: 1.1.1.1 (local to site)
Protocol: TCP
Port: 4000
aclID: 200 (passed to function)
Direction: Normal

Query:

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID="200" and a.ruleID = b.ruleId
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("1.1.1.1”) | inet_aton(dstmask)) and
(inet_aton(srcip) | inet_aton(srcmask) = inet_aton("99.99.99.99”) | inet_aton(srcmask)) and
(proto="ip" or (proto="tcp" and ((dst_qual="eq" and dst_bgn=4000) or (dst_qual="gt" and dst_bgn <=
4000) or (dst_qual="lt" and dst_bgn >= 4000 ) or (dst_qual = "range" and 4000 between dst_bgn and
dst_end)) )) and flags is NULL order by seqNum limit 1;
```

The reverse flow query would then be called with the same data defined, except the directional flag would be set to “reverse” and the aclID would presumably be different as well (201):

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID="201" and a.ruleID = b.ruleId
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("99.99.99.99") | inet_aton(dstmask)) and
(inet_aton(srcip) | inet_aton(srcmask) = inet_aton("1.1.1.1") | inet_aton(srcmask)) and (proto="ip"
or (proto="tcp" and ((srcp_qual="eq" and src_bgn=4000) or (src_qual="gt" and src_bgn <= 4000) or
(src_qual="lt" and src_bgn >= 4000 ) or (src_qual = "range" and 4000 between src_bgn and src_end)
or (dst_qual = "gt" and flags like "%estab%") or (src_qual is NULL and dst_qual is NULL and flags
like "%estab%") or (dst_qual = "gt" and dst_bgn <= 4000) or (dst_qual = "lt" and flags like
"%estab%")) )) and ((flags not like "%frag%") or (flags is NULL)) order by seqNum limit 1;
```

Example 2

Source IP address: 1.1.1.1. (internal to site
Destination IP address: 99.99.99.99 (external to site)
Protocol: UDP
Port: 1000
aclID: 300 (passed to function)
Direction: Normal Bi-Directional

Query:

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID="300" and a.ruleID = b.ruleId
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("99.99.99.99") | inet_aton(dstmask)) and
(inet_aton(srcip) | inet_aton(srcmask) = inet_aton("1.1.1.1") | inet_aton(srcmask)) and
(proto="ip" or (proto="udp" and ((dst_qual="eq" and dst_bgn=1000) or (dst_qual="gt" and dst_bgn <=
1000) or (dst_qual="lt" and dst_bgn >= 1000 ) or (dst_qual = "range" and 1000 between dst_bgn and
dst_end)) )) and flags is NULL order by seqNum limit 1;
```

Given that this is a bi-directional flow, the reverse will be checked, with aclID set to "301":

```
select b.*, a.seqNum from ruleDetails a, ruleFields b where a.aclID="301" and a.ruleID = b.ruleId
and (inet_aton(dstip) | inet_aton(dstmask) = inet_aton("1.1.1.1") | inet_aton(dstmask)) and
(inet_aton(srcip) | inet_aton(srcmask) = inet_aton("99.99.99.99") | inet_aton(srcmask)) and
(proto="ip" or (proto="udp" and ((src_qual="eq" and src_bgn=1000) or (src_qual="gt" and src_bgn <=
1000) or (src_qual="lt" and src_bgn >= 1000 ) or (src_qual = "range" and 1000 between src_bgn and
src_end) or (dst_qual = "gt" and dst_bgn <= 1000) or (dst_qual = "gt" and dst_bgn <= 1024)) )) and
((flags not like "%frag%") or (flags is NULL)) order by seqNum limit 1;
```

The result from each query will have its *action* field evaluated. If the action is "permit," then the packet will pass through the firewall and the function will return a value of "1." Otherwise, the action is "denied," in which case a value of "0" is returned. If no matches are found, then a value of "0" is returned since the general rule of firewalls is to implicitly deny anything not permitted.

Throughout the process, there are record-keeping variables $denied and $revDenied that were set to a value of "1" if the firewall query returned a negative response. Since it only takes one direction to be denied, a simple bitwise OR operation is performed on these two values. If either are set to "1," then the overall result is that the packet will be denied at some point, restricting the connection to be made. If both values are still set to their default of "0," then the connection is permitted.

VI. User Interface

With all the processing being done on the back-end, the front-end needs only a few simple user input fields. Basic input validation checks should be done to verify IP addresses, and additionally, if a transport protocol such as TCP or UDP is chosen, then a port should be specified. It is also convenient for the user if they can provide hostnames instead of IP addresses to add an additional level of translation. In the sample front-end shown in Figure 1 below, the input fields are collected using a standard HTML form.

After the user submits the information, the following page can display a variety of information. In the clearest form, a simple "Access Blocked" or "Access Permitted" is shown. To help with troubleshooting, more information can be displayed. In the example output shown in Figure 2 below, each router name along the path is displayed, including which firewall rule sets are checked and whether the flow was permitted or blocked by that rule set. This makes it easy for the user to see where exactly their traffic is being blocked, and reduces overall troubleshooting time. Additionally, the actual rule that gets matched

can be displayed to see which rule is blocking or permitting the traffic.

Figure 1. Sample HTML user interface front-end.

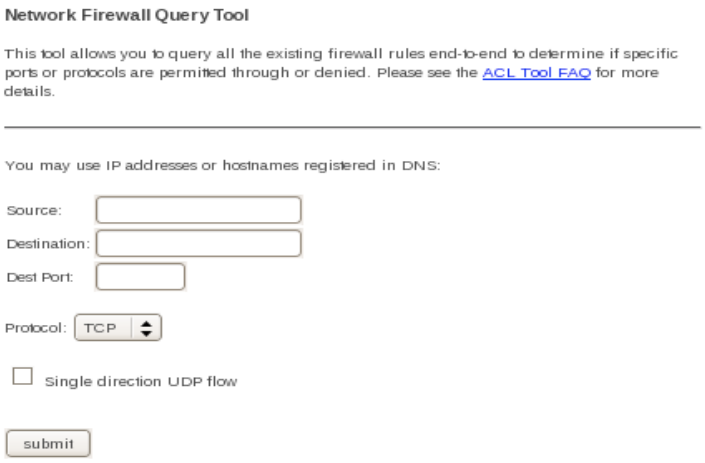
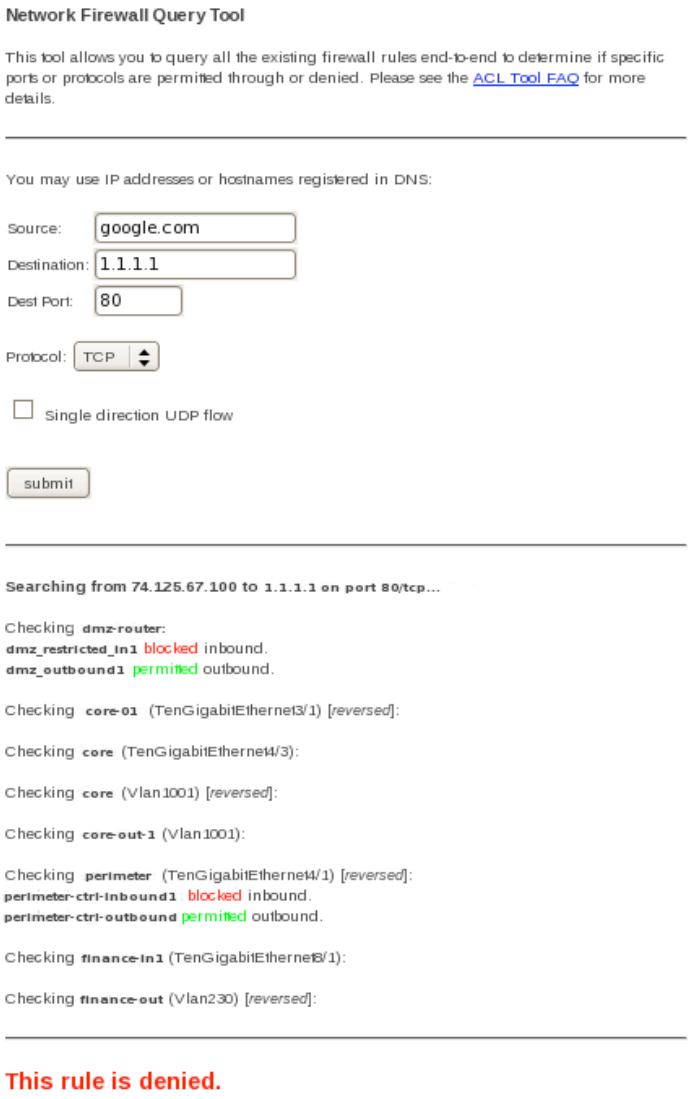


Figure 2. Sample output from user interface front-end.



It is important to keep your user-base in mind, though. If your users are on your internal network and are trusted, giving them this information may not be much of a security risk. However, if the people using this tool are anonymous outside users, then only the minimum amount of information necessary should be displayed. Alternatively, an authentication page can be used to verify users prior to giving them access to this tool.

Conclusion

The scalable and robust firewall policy query tool described in this paper can be used by many organizations that have to deal with a tangled web of access rules. It navigates that web, and tells you where it went and what it found, effectively eliminating the time-consuming manual search process. This abstraction also removes the need to understand and identify all the technical components, so that non-network-savvy users can also use the tool to troubleshoot and identify access issues.

Software does not yet exist in the commercial market or open source community to provide this functionality in a simple web front-end, accessible to all levels of staff or even anonymous users.

The time it took to develop and deploy this operationally was nearly equivalent to the time spent evaluating commercial products, which ended up not meeting our requirements. Additionally, there was no cost associated with this project (other than personnel time required for development) since all software used was open-source freeware and could run on existing hardware already used for other services.